

Eddy Fromentin and Michel Raynal*

IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France

Received January 25, 1996; revised April 4, 1997

A consistent observation of a given distributed computation is a sequence of global states that could be produced by executing that computation on a monoprocessor system. Therefore a distributed execution generally accepts several consistent observations. This paper investigates global states shared by all such observations. A necessary and sufficient condition characterizing these states is first given. Then, an algorithm that computes shared global states is described.

© 1997 Academic Press

1. INTRODUCTION

Since Lamport's seminal paper [10], the set of events produced by an execution of an asynchronous distributed program is modeled as a partial order. Due to the asynchronous nature of the underlying support (no common physical clock, no shared memory and arbitrary transfer delays), any consistent external observer of such a distributed execution can only see a sequence including all events and respecting their partial order [14]. Using this sequence of events the observer can construct, starting from the initial state of the computation, a sequence of global states through which the computation might have progressed; such a sequence of global states constitutes an *observation* of the distributed computation [1, 14]. One important question is then: Are there global states *shared* by all possible observations of a distributed computation? When there are such global states, they are independent of observers.

This paper presents a characterization of shared global states. It is composed of three parts. Section 2 introduces a model for distributed computations. Section 3 gives a necessary and sufficient condition for a global state to be shared. Section 4 describes an algorithm to detect shared global states.

2. DISTRIBUTED COMPUTATIONS

2.1. Distributed Programs

A distributed program is composed of n sequential processes P_1, \dots, P_n which communicate and synchronize

only by message passing. These distributed programs are executed by an underlying system composed of processors that can exchange messages. Each processor has a local memory. Neither shared memory nor a global clock is available. Messages are exchanged through reliable, not necessarily FIFO, channels. Transmission delays are finite but unpredictable.

2.2. Distributed Computations

2.2.1. Primitive Events. Execution of a process P_i produces a sequence of *primitive events*. These events result from the execution of program statements. A primitive event may be either *internal* (causing only a change to local variables) or it may involve communication (*send* or *receive* event) [10]. This sequence is usually called the history H_i of P_i : $H_i = e_i^1 e_i^2 e_i^3 \dots e_i^x \dots$ where e_i^x is the x th primitive event produced by P_i .

Let H be the set of all the events produced by an execution of a distributed program and let \xrightarrow{e} be a binary relation on these primitive events defined as Lamport's *causal precedence* relation [10] (which is antisymmetric and transitive) extended to be reflexive (so \xrightarrow{e} defines a partial order):

$$e_i^x \xrightarrow{e} e_j^y \Leftrightarrow \begin{cases} e_i^x = e_j^y \\ \text{or} \\ i = j \text{ and } x + 1 = y \\ \text{or} \\ e_i^x \text{ is the send of a message } m \text{ and } e_j^y \text{ its receive event} \\ \text{or} \\ \exists e_k^z \text{ such that } e_i^x \xrightarrow{e} e_k^z \text{ and } e_k^z \xrightarrow{e} e_j^y \end{cases}$$

When considering primitive events, a distributed execution is modeled as a partially ordered set (poset) $\hat{H} = (H, \xrightarrow{e})$ [10]. This poset represents the computation at some abstraction level that we call the *primitive level*. By definition this level comprises all communication events and internal events. Figure 1 displays a distributed computation at some primitive level in the classical space-time diagram where events are denoted by black and white circles and messages by arrows.

* E-mail: {fromentin,raynal}@irisa.fr.

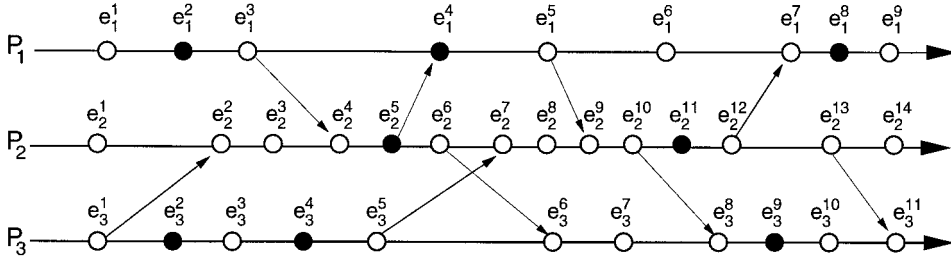


FIG. 1. A distributed execution at primitive level.

2.2.2. Relevant Events. At some abstraction level only some events of a distributed computation are relevant [5, 12]. For example, in Fig. 1 only those events denoted by a black circle are relevant. Let R be the set of these relevant events.¹ The poset $\hat{R} = (R, \xrightarrow{c})$ constitutes an abstraction of the distributed computation at the abstraction level considered. Interestingly, as \hat{R} is a subposet of \hat{H} , thanks to the transitivity of \xrightarrow{c} , \hat{R} inherits causal precedence induced by communication events, even if communication events are not relevant. In the following, we will only consider the poset \hat{R} .

2.3. Local and Global States

2.3.1. Definitions. Define R_i as the history of P_i including only its relevant events: $R_i = r_i^1 r_i^2 \dots r_i^x \dots$ (where $r_i^x (x > 0)$ is the x th, relevant event of P_i). Let s_i^0 be the initial local state of P_i . The event r_i^x provokes the change from local state s_i^{x-1} to local state s_i^x (next(s_i^{x-1}) will be synonymous with s_i^x and prev(s_i^x) will be synonymous with s_i^{x-1}). Informally, a local state s_i^x spans from event r_i^x till r_i^{x+1} . Figure 2 shows local states of the distributed computation of Fig. 1 from which irrelevant events (white circles) have been eliminated. If process P_i terminates, s_i^{last} denotes its last local state. A global state Σ of a distributed computation is a set of n local states, one from each process: $\Sigma = (s_1, \dots, s_i, \dots, s_n)$.

2.3.2. Precedence Relation on Local States. The set of local states of a distributed computation \hat{R} is partially ordered by an irreflexive partial order relation called *local state precedence*, denoted \xrightarrow{s} . Informally, $s_i \xrightarrow{s} s_j$ means that s_i was no longer existing when s_j began to exist. Formally, local state precedence is defined in the following way:

$$s_i^x \xrightarrow{s} s_j^y \Leftrightarrow r_i^{x+1} \xrightarrow{s} r_j^y.$$

Two distinct local states s_i and s_j are said to be concurrent [1, 3, 14], denoted $s_i \parallel s_j$, if and only if $\neg(s_i \xrightarrow{s} s_j)$ and

¹ Here are two examples illustrating relevant events. In the detection of unstable properties [4], only updates of some variables are meaningful to the user (e.g., if one is interested in the detection of the global predicate $x_1 + x_2 + x_3 + \dots + x_n < k$ where x_i is a local variable of P_i , relevant events of P_i are modifications of x_i). In defining and testing the consistency of recovery lines in backward recovery, only events producing local checkpoints are relevant [17].

$\neg(s_j \xrightarrow{s} s_i)$. A global state $\Sigma = (s_1, \dots, s_i, \dots, s_n)$ is consistent iff $\forall(i, j): i \neq j :: s_i \parallel s_j$.

3. SHARED GLOBAL STATES

3.1. The Lattice of Global States

The set of all the consistent global states of a distributed computation $\hat{R} = (R, \xrightarrow{c})$ forms a lattice $\mathcal{L}(\hat{R})$ whose minimal (resp. maximal) element is the initial (resp. final) global state $\Sigma^0 = (s_1^0, \dots, s_i^0, \dots, s_n^0)$ (resp. $\Sigma^{\text{last}} = (s_1^{\text{last}}, \dots, s_i^{\text{last}}, \dots, s_n^{\text{last}})$) [13]. There is an edge from a vertex $\Sigma = (s_1, \dots, s_i, \dots, s_n)$ to a vertex $\Sigma' = (s_1, \dots, \text{next}(s_i), \dots, s_n)$ if and only if there is an event r_i of P_i that can be produced in global state Σ (so the event r_i entails the local state change from s_i to next(s_i)). Figure 3 shows the lattice $\mathcal{L}(\hat{R})$ associated with the distributed computation \hat{R} depicted in Fig. 2.

Informally, a sequential observation of a distributed computation \hat{R} represents a view of \hat{R} that an external sequential observer could have [14]. More formally, we define a sequential observation O as a sequence $\Sigma^0 \Sigma^1 \Sigma^2 \dots \Sigma^{k-1} \Sigma^k \Sigma^{k+1} \dots$ of consistent global states such that:

- there exists a sequence $O_r = r^1 r^2 \dots r^k \dots$ of events which is a linear extension of \hat{R} ,
- let Σ^{k-1} be $(s_1, \dots, s_i, \dots, s_n)$ and P_i be the process that produced r^k . Then $\Sigma^k = (s_1, \dots, \text{next}(s_i), \dots, s_n)$ (and P_i entered next(s_i) by executing r^k).

A sequential observation corresponds to a path in the lattice of global states. As shown in [14] there is an one-to-one correspondence between all the possible observations of a distributed computation and all the paths of the lattice (the interested reader can find more details about observations in [1, 14]).³

² The final global state Σ^{last} exists only if all processes of the distributed computation terminate.

³ Alternatively, an observation can be defined as a sequence O_r of events which is a linear extension of \hat{R} [1, 14]. As shown by the lattice $\mathcal{L}(\hat{R})$, both definitions are equivalent: one is event-based while the other is state-based.

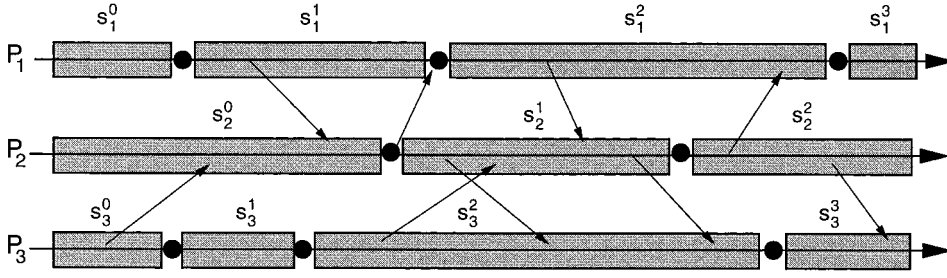


FIG. 2. Local states of a distributed execution.

3.2. Shared Global States

3.2.1. Definition. Let us consider a distributed computation \hat{R} and its associated lattice $\mathcal{L}(\hat{R})$. A global state is *shared* if it belongs to all the observations of the distributed computation. For example, in the lattice depicted in Fig. 3 the global state $\Sigma = (s_1^2, s_2^1, s_3^2)$ is a shared global state.

The number of shared global states depends on both the computation and the abstraction level. So according to the abstraction level defining relevant events, it is possible that very few global states of a computation be shared (in the worst case, only the initial global state and the final one—if it exists—are shared).

3.2.2. A Necessary and Sufficient Condition

THEOREM (SGS). Let $\Sigma = (s_1, \dots, s_i, \dots, s_n)$ be a global state. We have

$$\Sigma \text{ is shared} \Leftrightarrow$$

$$\forall(i, j) :: (\text{prev}(s_i) \xrightarrow{s} \text{next}(s_j) \text{ or } s_j = s_j^{\text{last}} \text{ or } s_i = s_i^0).$$

Proof. The following notations are used in the proof.

- r_i and $n-r_j$ denote the events that created the local states s_i and $\text{next}(s_j)$, respectively (see Fig. 4).

- Let $\Sigma = (s_1, \dots, s_i, \dots, s_j, \dots, s_n)$ be a consistent global state. Considering an observation O , we use the following global states defined with respect to local states of Σ :

- $F(\Sigma, i)$ denotes the first consistent global state of O whose i th component is s_i (the same as the one of Σ).

- $FN(\Sigma, j)$ denotes the first consistent global state of O whose j th component is $\text{next}(s_j)$.

The proof supposes that processes have infinite behaviors; it can be easily extended to finite behaviors. Moreover, to simplify reasoning we associate with each observation $O = \Sigma^0 \Sigma^1 \dots \Sigma^k \Sigma^{k+1} \dots$ the corresponding sequence of events $O_r = r^1 r^2 \dots r^k r^{k+1} \dots$ which produced these global states.

if part

Let $\Sigma = (s_1, \dots, s_i, \dots, s_j, \dots, s_n)$ be a shared global state and O be any observation.

1. O includes:

- Σ (as it is shared);
- $\forall j :: FN(\Sigma, j) = (\dots, \text{next}(s_j), \dots)$ (as O is generated by a linear extension of \hat{R} which includes $n-r_j$).

2. As $\forall j :: P_j$ is in s_j (by executing r_j) before entering $\text{next}(s_j)$ (by executing $n-r_j$), we conclude that $\Sigma = (s_1, \dots, s_i, \dots, s_j, \dots, s_n)$ appears in O before $FN(\Sigma, j)$, $\forall j$.

3. From (2) we conclude that each event r_i ($i = 1, \dots, n$) appears before each event $n-r_j$ ($j = 1, \dots, n$) in O_r .

4. As Σ is shared (hypothesis), point (3) is true for all the observations, i.e., in all the linear extensions O_r of \hat{R} , we have $\forall(i, j) :: r_i$ appears before $n-r_j$.

5. From Szpilrajn's theorem [15] (which states that the intersection of all linear extensions of a partial order—here $\hat{R} = (R, \xrightarrow{e})$ —is precisely this partial order) it follows that: $\forall(i, j) :: r_i \xrightarrow{e} n-r_j$.

6. From the definitions of events r_i and $n-r_j$, the definition of the local state precedence and point (5), it follows that $\forall(i, j) :: \text{prev}(s_i) \xrightarrow{s} \text{next}(s_j)$.

only if part

Let $\Sigma = (s_1, \dots, s_i, \dots, s_j, \dots, s_n)$ be a global state such that $\forall(i, j) :: \text{prev}(s_i) \xrightarrow{s} \text{next}(s_j)$.

1. In terms of events the previous relation is equivalent to: $\forall(i, j) :: r_i \xrightarrow{e} n-r_j$.

2. Σ is a consistent global state. Suppose that it is not; then $\exists(i, j) :: s_j \xrightarrow{s} s_i$, i.e., $n-r_j \xrightarrow{e} r_i$ which contradicts point (1).

3. Let O be an arbitrary observation and consider for each pair (i, j) the two global states $F(\Sigma, i)$ and $FN(\Sigma, j)$. From point (1), namely $\forall(i, j) :: r_i \xrightarrow{e} n-r_j$, it follows that $\forall(i, j) :: F(\Sigma, i)$ appears before $FN(\Sigma, j)$ in O .

4. From (3), O is such that:

$$O = \Sigma^0 \dots F(\Sigma, i_1) \dots F(\Sigma, i_x) \dots F(\Sigma, i_n) FN(\Sigma, j_1) \dots FN(\Sigma, j_x) \dots FN(\Sigma, j_n) \dots$$

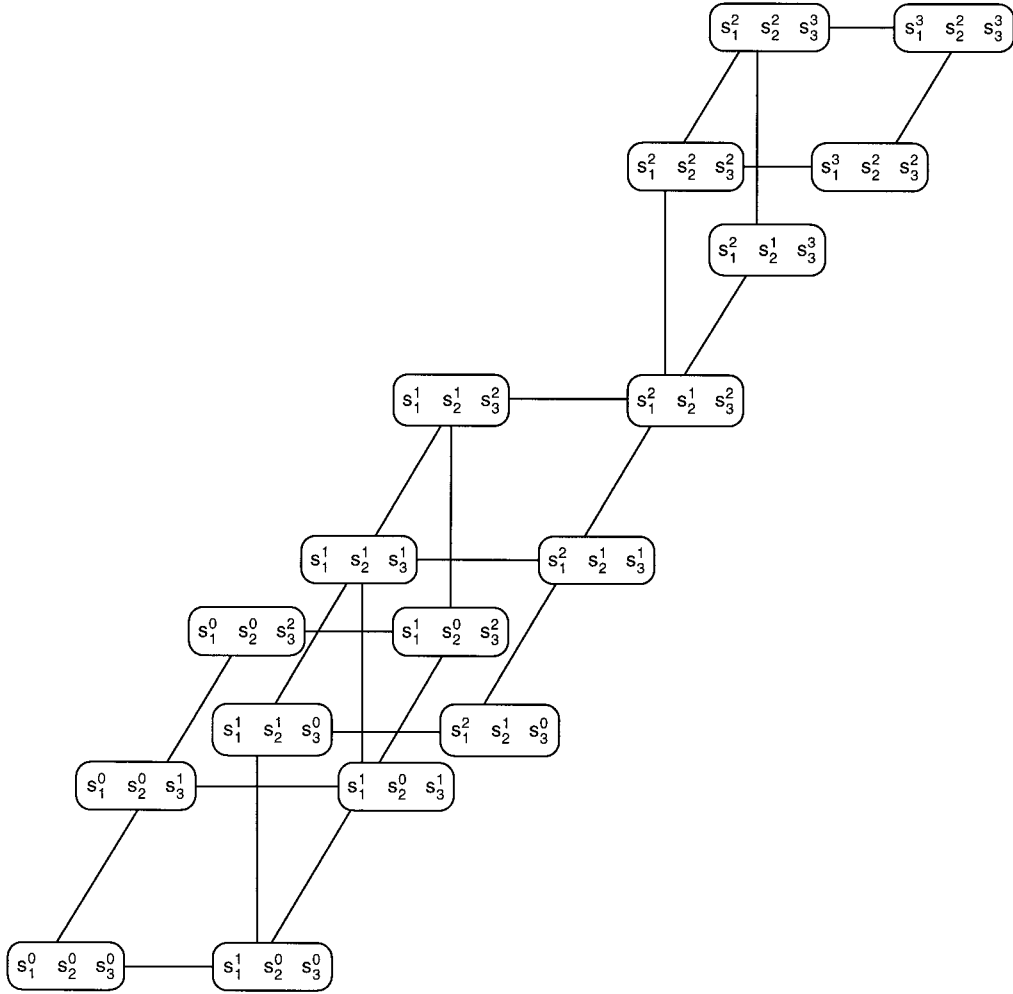


FIG. 3. Lattice associated with the distributed computation of Fig. 2.

with (i_1, \dots, i_n) and (j_1, \dots, j_n) being permutations of $(1, \dots, n)$. It follows that $F(\Sigma, i_n) = (s_1, \dots, s_{i_n}, \dots, s_n) = \Sigma$, and then Σ belongs to O .

5. As O was an arbitrary observation, points (3) and (4) apply to all the observations. It follows that Σ is shared. ■

3.2.3. Interest of Shared Global States. In addition to their conceptual interest, shared global states present a practical interest as their determination can be done at low cost (see Section 4), i.e., without building the lattice $\mathcal{L}(\hat{R})$ of consistent global states. This reveals to be particularly attractive when detecting predicates such as $DEF \Phi$ [4]. Let Φ be a predicate on global states. A distributed computation \hat{R} satisfies $DEF \Phi(\hat{R} \models DEF \Phi)$ if, and only if,

each observation O_x of \hat{R} includes a global state Σ_x such that $\Sigma_x \models \Phi$. As Φ can be any predicate on global states, classical algorithms to detect $DEF \Phi$ are based on a traversal of the entire lattice of global states [1, 2, 4]. Shared global states can be seen as a cheap heuristic alternative to limit the search space as:

$$(\exists \Sigma \text{ shared: } \Sigma \models \Phi) \Rightarrow (\hat{R} \models DEF \Phi).$$

Let us consider the special case where Φ is a conjunction of local predicates, i.e., $\Phi = \bigwedge_{i=1}^n LP_i$ (LP_i being a predicate on local variables of P_i). Let $\Sigma = (s_1, \dots, s_n)$; by definition $\Sigma \models \Phi$ iff $\bigwedge_i (s_i \models LP_i)$. Moreover, let us consider an abstraction level where P_i produces a relevant event (and enters a new local state) each time LP_i changes its value. Expressed with the shared global states abstraction, results of [9, 16] can be reformulated at a more abstract level as [7]:

$$(\hat{R} \models DEF \left(\bigwedge_i LP_i \right)) \Leftrightarrow (\exists \Sigma \text{ shared: } \Sigma \models \left(\bigwedge_i LP_i \right)).$$

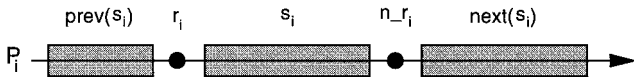


FIG. 4. Notations for event names.

4. DETECTING SHARED GLOBAL STATES

The SGS Theorem allows us to design an algorithm that detects shared global states. A FIFO channel is added between each process and a monitor M . Each time a new local state begins, P_i sends to M a control message composed of its current local state plus its vector timestamp.⁴ The monitor is equipped with n queues: Q_i stores incoming messages from process P_i .

The protocol executed by the monitor M is an adaptation of an algorithm defined by Garg [8] to detect a largest antichain (here, a n -tuple of local states satisfying the SGS Theorem) in a partially ordered set given its decomposition into its chains (here, the sequences of control messages received from each P_i and stored in queues Q_i). The protocol is described in the following subsections. (It can be decentralized using the technique described in [9].) Let k_i be the number of local states of process P_i and $K = \max_i(k_i)$. Section 4.4 shows that $O(n^3K)$ is an upperbound of the number of comparisons of integers of the algorithm.⁵

4.1. Underlying Principles

In order to detect shared global states, Garg's algorithm [8] is adapted in the following way. Q_i is the sequence of timestamped local states received in FIFO order from P_i ; $\text{head}(Q_i)$ denotes the first local state of Q_i ; $\text{tail}(Q_i)$ denotes the sequence Q_i without its first element; $\text{next}^*(s_i)$ denotes any local successor of s_i including s_i itself.

Deciding whether two local states are related by a precedence relation can be done in a classical way by using vector timestamps.⁶ To make the algorithm easier to understand we suppose the queues Q_i have been filled up by processes. This version can easily be adapted to work on the fly, with processes filling their queues as they progress.

In Garg's algorithm the heads of the queues are checked to see if they form a global state (a largest antichain). Its adaptation for detecting shared global states is based on the following observations:

1. Let $\Sigma = (..., s_i, ..., s_j, ...)$ be the global state under consideration candidate to be shared. if $\neg(\text{prev}(s_i) \xrightarrow{s} \text{next}(s_j))$ we can conclude any global state $\Sigma' = (..., \text{next}^*(s_i), ..., s_j, ...)$ is not shared. So in that case s_j is no longer considered and the algorithm considers the global state $\Sigma'' = (..., s_i, ..., \text{next}(s_j), ...)$ as a candidate to be shared. This observation permits us to redefine appropriately the

head of the queues (with the auxiliary variable *changed* in the algorithm).

2. After a shared global state $\Sigma = (s_1, s_2, ..., s_n)$ has been found, the next candidate Σ' to be shared is defined in the following way (in order not to miss shared global states): Σ' is a consistent global state $(s'_1, s'_2, ..., s'_n)$ that is an immediate successor of Σ in the lattice, i.e., $\exists k$ with $(\forall i \neq k, s'_i = s_i)$ and $(s'_k = \text{next}(s_k))$.

4.2. The Algorithm

The algorithm is described in Fig. 5. For any queue Q_i , s_i is synonymous with $\text{head}(Q_i)$, and $\text{next}(s_i)$ is synonymous with $\text{head}(\text{tail}(Q_i))$.

4.3. Safety and Liveness

SAFETY (The Detection Is Consistent). *If the algorithm claims Σ is shared, then it is.*

Proof. The proof follows directly from conditions tested at lines 1 and 2: if, at line 3, a global state Σ is declared shared, it necessarily satisfies condition SGS theorem. ■

LIVENESS. *If a global state is shared, then the algorithm detects it.*

Proof. Consider that the algorithm is in its initial state or just after a global state has been declared shared. Suppose a shared global state $\Sigma' = (s'_1, ..., s'_n)$ exists, and it is the first, in the lattice, of next shared global states. Suppose also that the algorithm is checking at lines 1 and 2 a global state $\Sigma'' = (s''_1, ..., s''_n)$ such that Σ'' is a (not necessarily immediate) successor of Σ' in the lattice (such a Σ'' does exist; at worst it is $(s_1^{\text{last}}, s_2^{\text{last}}, ..., s_n^{\text{last}})$). Finally, suppose that Σ' has not been found by the algorithm. We show there is a contradiction.

All elements of any queue are examined by the algorithm. So, at some time t_1 , we have $s'_i = \text{head}(Q_i)$ and s'_i is removed from Q_i without declaring Σ' shared. Consequently, there exists a head of some queue Q_j (let it be $\tau_j = \text{head}(Q_j)$), such that $\neg(\text{prev}(\tau_j) \xrightarrow{s} \text{next}(s'_i))$. We can conclude that $\tau_j \neq s'_j$ (as Σ' is shared, $\text{prev}(s'_j) \xrightarrow{s} \text{next}(s'_i)$) and $s'_j \xrightarrow{s} \tau_j$ (else, we would have $\text{prev}(\tau_j) \xrightarrow{s} s'_j$ —as these two local states are from the same P_j —and $\text{prev}(s'_j) \xrightarrow{s} \text{next}(s'_i)$ —as Σ' is shared—and by transitivity we would have $\text{prev}(\tau_j) \xrightarrow{s} \text{next}(s'_i)$).

As $\text{head}(Q_j) = \tau_j$, the algorithm has already eliminated s'_j from Q_j at some time $t_2 (t_2 < t_1)$. Consider now the algorithm at t_2 : at that time there existed $\tau_h = \text{head}(Q_h)$ that provoked the elimination of s'_j from Q_j (same reasoning as before). By induction on the number of queues it follows that at $t_{n-1} (t_{n-1} < t_{n-2} < \dots < t_1)$ we had:

$$\exists k: \begin{cases} \text{head}(Q_k) = \tau_k \text{ with } s'_k \xrightarrow{s} \tau_k, \text{ and} \\ \text{head}(Q_i) = s_i \text{ with } s_i \xrightarrow{s} s'_i \vee s_i = s'_i \quad (\text{for } i \neq k). \end{cases}$$

⁴ A vector timestamp is composed of n integers [6].

⁵ $O(n^2K)$ is the time complexity if we search only the first shared global state.

⁶ Detecting whether $s_i \xrightarrow{s} s_j$ is done by comparing the vector timestamp associated with s_i with the vector timestamp associated with s_j . This requires n integer comparisons in the worst case. Detecting $s_i \parallel s_j$ can be done with only two integer comparisons [6, 14]. These numbers of integer comparisons will be used in Section 4.4 to compute time complexity.

```

changed := {1, 2, ..., n};
while  $\exists i : s_i \neq s_i^{\text{last}}$  do
    newchanged :=  $\emptyset$ ;
    % evaluation of the SGS Theorem on  $\Sigma = (s_1, \dots, s_i, \dots, s_j, \dots, s_n)$ 
     $\forall i \in \text{changed}, j \in \{1, 2, \dots, n\}$  do
        (1) if  $\neg(s_j = s_j^{\text{last}}$  or  $s_i = s_i^0$  or  $\text{prev}(s_i) \xrightarrow{s} \text{next}(s_j))$  then
            newchanged := newchanged  $\cup \{j\}$ 
        fi;
        (2) if  $\neg(s_i = s_i^{\text{last}}$  or  $s_j = s_j^0$  or  $\text{prev}(s_j) \xrightarrow{s} \text{next}(s_i))$  then
            newchanged := newchanged  $\cup \{i\}$ 
        fi;
    od;
    (3) if newchanged =  $\emptyset$  then
        (4)  $\Sigma = (s_1, \dots, s_i, \dots, s_j, \dots, s_n)$  is shared;
        (5) let  $k$  such that  $\Sigma' = (s'_1, \dots, s'_k, \dots, s'_n)$  is a consistent global state
            with  $(\forall i \neq k, s_i = s'_i)$  and  $(s'_k = \text{next}(s_k))$ ,
            newchanged :=  $\{k\}$ ;
    fi;
    changed := newchanged;
     $\forall i \in \text{changed} : Q_i := \text{tail}(Q_i)$ ;
od;
```

FIG. 5. Computing shared global states.

So, s'_k has been eliminated from its queue Q_k at $t_n (t_n < t_{n-1})$ at lines 1 and 2 by a local state s_j such that $s_j \xrightarrow{s} s'_j \vee s_j = s'_j$ (i.e., we had at $t_n : \neg(\text{prev}(s_j) \xrightarrow{s} \text{next}(s'_k))$). This is impossible because Σ' is shared (we cannot have $(s_j \xrightarrow{s} s'_j \vee s_j = s'_j) \wedge \neg(\text{prev}(s_j) \xrightarrow{s} \text{next}(s'_k))$). Hence the contradiction. It follows that Σ' has not been missed. ■

4.4. Time Complexity

Let k_i be the number of local states (including s_i^0 and s_i^{last}) of process P_i and $K = \max_i(k_i)$.

If we eliminate the statement **if** $\text{newchanged} = \emptyset$ **then** ... **fi** we obtain an algorithm whose structure is the same as Garg's one. Garg showed, in [8], that the time complexity of his algorithm is $O(n^2K)$ comparisons, each comparison being on two integers.

Consider now the algorithm without the loop including lines 1 and 2. To advance the appropriate queue Q_k the algorithm has to find a consistent global state Σ' immediate successor of Σ . Obtaining such a global state $\Sigma' = (s_1, \dots, \text{next}(s_k), \dots, s_n)$ requires at most $O(n^2)$ comparisons of integers ($2(n-1)$ comparisons to test $\text{next}(s_k) \parallel s_i$ for $i \neq k$ and, in the worst case, finding the appropriate k requires n such sets of comparisons). Such tests are done each time a shared global state is found. $k_1 + k_2 + \dots + k_n$ constitutes an upper bound on the number of shared global states (all elements of queues are examined without never backtracking). So the second part of the algorithm is upper bounded by $O(n^2 \Sigma k_i)$.

Consequently $O(n^3K)$ constitutes an upper bound on the number of integer comparisons needed by the algorithm.

5. CONCLUDING REMARKS

This paper has introduced the concept of shared global state for distributed computations. Such states, defined with respect to an abstraction level, are not associated with particular observations: they are seen by all observers. A necessary and sufficient condition characterizing these global states has been given. An algorithm for on-the-fly detection of shared global states of a distributed computation has been designed by combining the algorithm described in [8] and the previous condition.

In [11] Lee and Davidson solve the generalized rendezvous problem in a real-time context. Such a rendezvous involves several processes and each of them specifies a deadline for its involvement in the rendezvous; for each process, its deadline defines a *real-time interval* (which begins at the time it wants to participate in the rendezvous and ends at its deadline). The generalized rendezvous is possible if the intersection of all these real-time intervals is not empty. Our necessary and sufficient condition for a global state to be shared expresses a similar notion with respect to logical time.

ACKNOWLEDGMENT

The authors are grateful to the referee whose comments helped improve the content of the paper.

REFERENCES

1. Ö. Babaoğlu and K. Marzullo, Consistent global states of distributed systems: Fundamental concepts and mechanisms, in "Distributed Systems" Frontier Series (S. J. Mullender, Ed.), Chap. 4, pp. 55–93, ACM Press, 1993.

2. Ö. Babaoğlu, E. Fromentin, and M. Raynal, A unified framework for the specification and run-time detection of dynamic properties in distributed computations, *J. Systems and Software* **3** No. 33 (1996), 287–298.
3. K. M. Chandy and L. Lamport, Distributed snapshots: Determining global states of distributed systems, *ACM Trans. Comput. Systems* **3** No. 1 (1985), 63–75.
4. R. Cooper and K. Marzullo, Consistent detection of global predicates, in “Proc. ACM/ONR Workshop on Parallel and Distributed Debugging, Santa Cruz, CA, May 1991,” pp. 167–174.
5. Cl. Diehl, Cl. Jard, and J. X. Rampon, Reachability analysis on distributed executions, in “Proc. Theory and Practice of Software Development” TAPSOFT, Lecture Notes in Computer Science, Vol. 668, pp. 629–643, Springer-Verlag, New York/Berlin, April 1993.
6. J. Fidge, Timestamps in message-passing systems that preserve the partial ordering, in “Proc. 11th Australian Computer Science Conference, February 1988,” pp. 55–66.
7. E. Fromentin and M. Raynal, Characterizing and detecting the global states seen by all the observers of a distributed computation, in “Proc. 15th IEEE International Conference on Distributed Computing Systems, Vancouver, June 1995,” pp. 431–438.
8. V. K. Garg, Some optimal algorithms for decomposed partially ordered sets, *Inform. Process. Lett.* **44** (1992), 39–43.
9. V. K. Garg and B. Waldecker, Detection of strong unstable predicates in distributed programs, *IEEE Trans. Parallel and Distributed Systems* **7** No. 12 (1996), 1323–1333.
10. L. Lamport, Time, clocks and the ordering of events in a distributed system, *Comm. ACM* **21**, No. 7 (1978), 558–565.
11. I. Lee and S. B. Davidson, Adding time to synchronous process communication, *IEEE Trans. Computers* **36**, No. 8 (1987), 941–948.
12. K. Marzullo and L. Sabel, Efficient detection of a class of stable properties, *Distrib. Comput.* **8**, No. 2 (1994), 81–91.
13. F. Mattern, Virtual time and global states of distributed systems, in “Proc. Workshop on Parallel and Distributed Algorithms” (Cosnard, Quinton, Raynal, and Roberts, Eds.), pp. 215–226, North-Holland, Amsterdam, October 1988.
14. R. Schwarz and F. Mattern, Detecting causal relationships in distributed computations: In search of the holy grail, *Distributed Comput.* **7** No. 3 (1994), 149–174.
15. E. Szpilrajn, Sur l’extension de l’ordre partiel, *Fund. Math.* **16** (1930), 386–389.
16. S. Venkatesan and B. Dathan, Testing and debugging distributed programs using global predicates, in “Proc. 13th Annual Allerton Conference on Communication, Control, and Computing, 1992,” pp. 137–146.
17. Y. M. Wang, A. Lowry, and W. K. Fuchs, Consistent global checkpoints based on direct dependency tracking, *Inform. Process. Lett.* **50** (1994), 223–230.